Filtered Ranked Segment Search for Visual Exploration of Large Time Series Data

UI and Database Operators in a modern TS Database
CS2270 Project
May 6, 2019

Mary McGrath and Junjay Tan

Motivation: Anomaly Detection

Detectors often have *intermediate outputs*

Thresholds on these outputs are typically used to flag anomalies

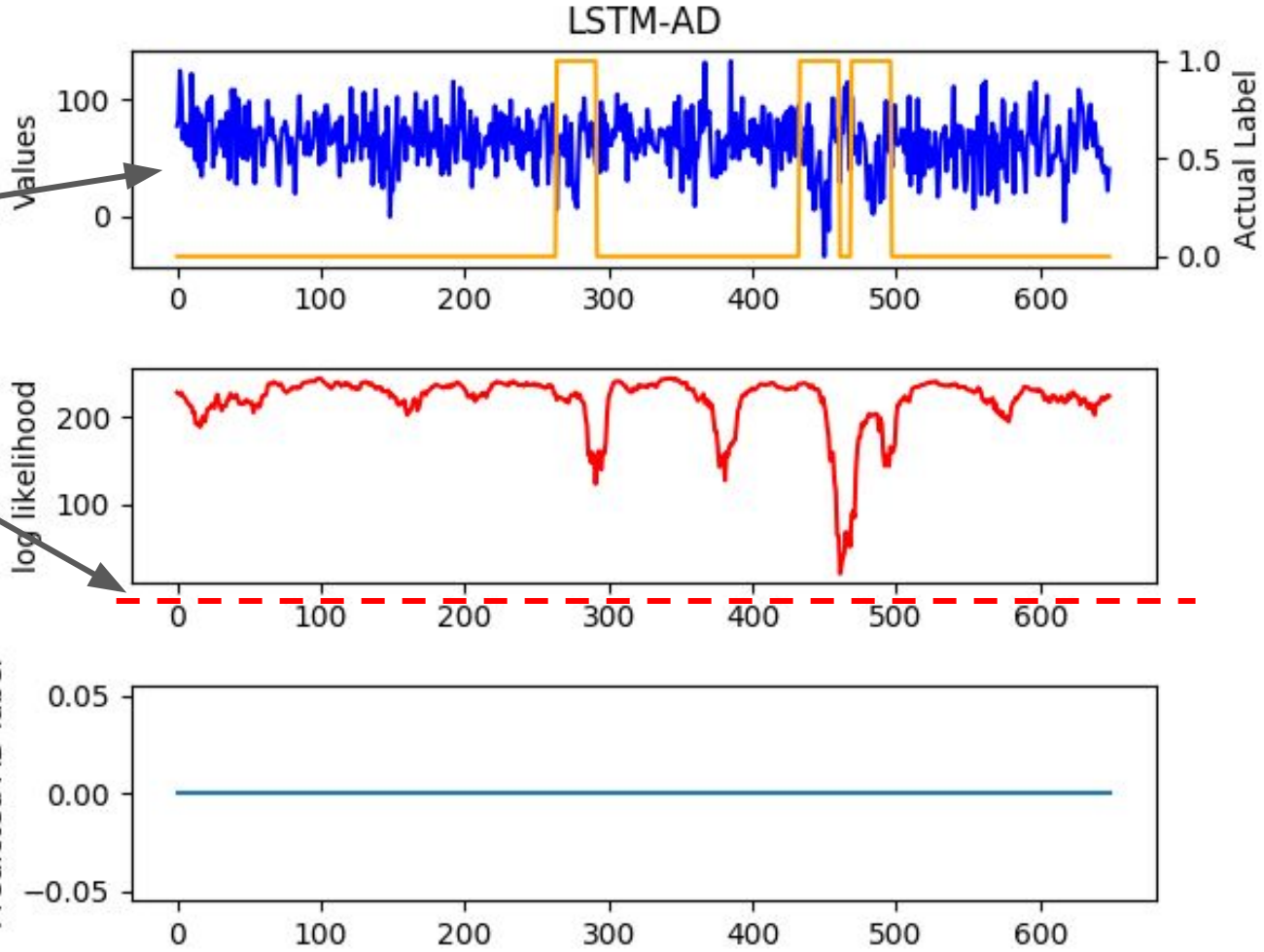
Values just below the threshold or in the anomalous ranges can be informative to inspect further

Example: LSTM-AD on Numenta Data (Minneapolis real-time traffic)

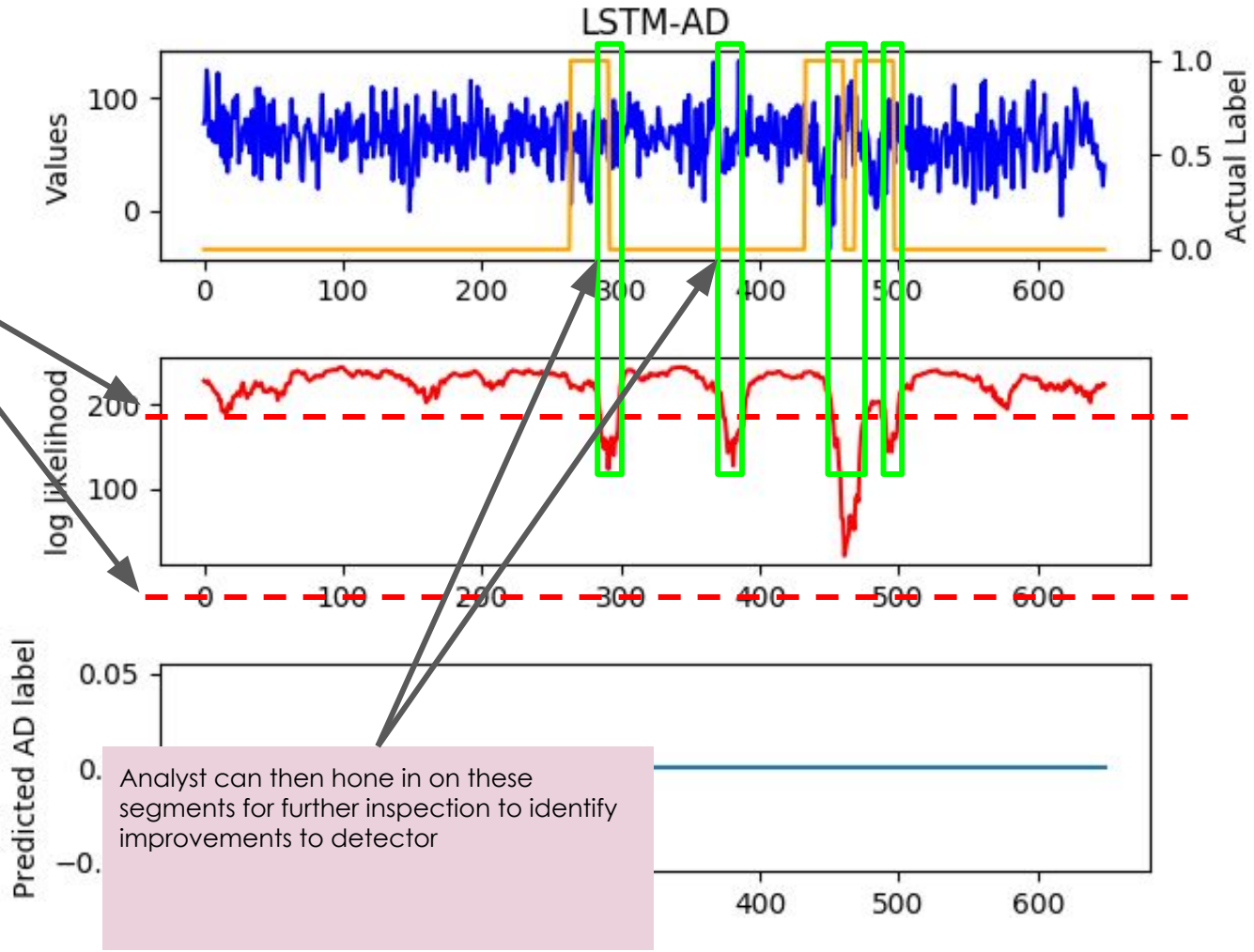
Raw TS data with anomalous regions overlaid

Detector output (likelihood value): Threshold determined from test data is below this value.

Therefore, no anomalies predicted!



Examining segments in this likelihood range is valuable for visually identifying anomalies in the raw data



Analyst can then hone in on these segments for further inspection to identify improvements to detector

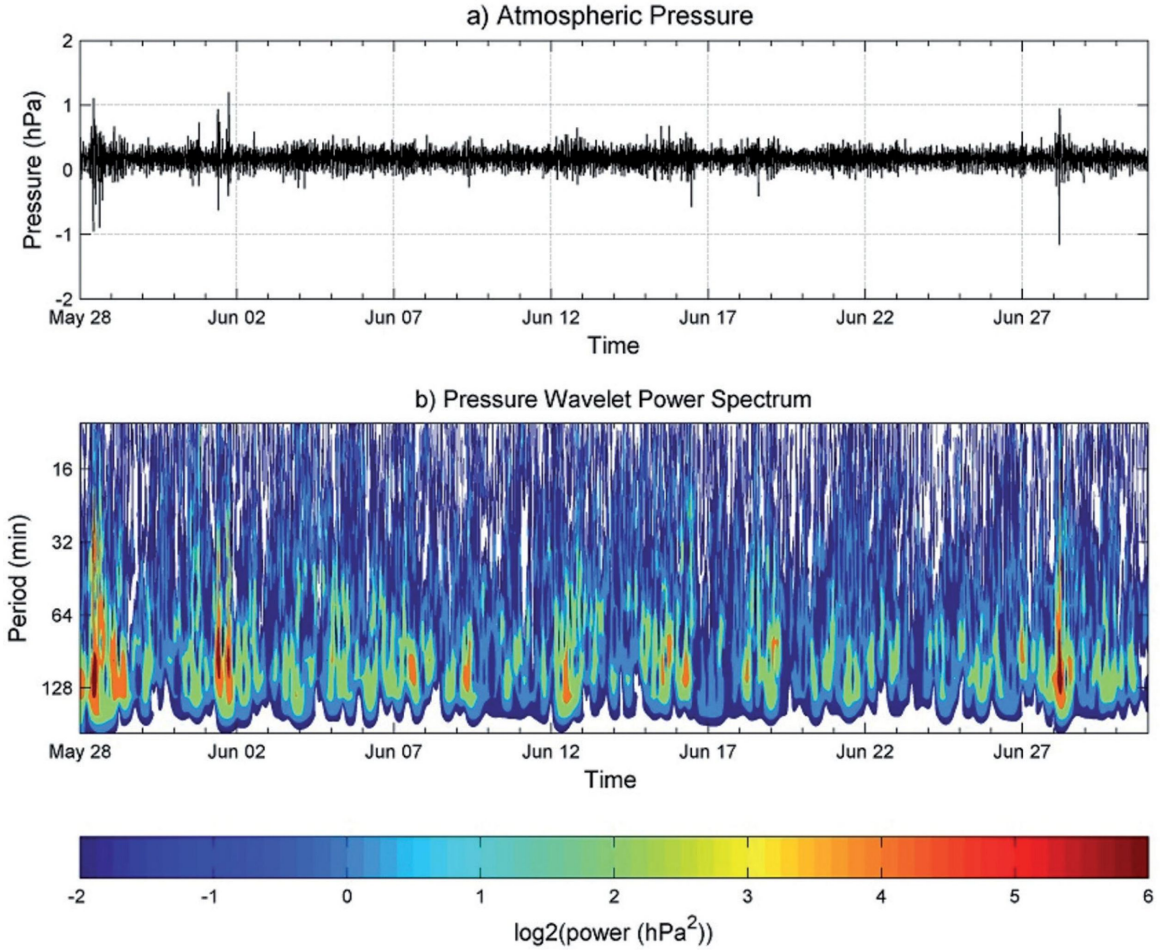
Motivation: Wavelet Power Spectrum Analysis

At each localized time (window), calculate signal power by frequency
→ Output is a new time series!

Brown Neuromotion Lab: Interested in where some frequency power exceeds the average power for initial exploration

WPSA typically done as a batch process (hours), with outputs loaded into data store for analysis

Earth Science Example



Each **time point** (horizontal axis) consists of multiple **frequencies** (vertical axis), each of which has a **power spectrum value**

L Perez, D Walter. Spectral variability in high frequency in sea level and atmospheric pressure on Buenos Aires Coast, Argentina. Brazilian Journal of Oceanography. 2017.

Definitions

"Find **segments** (i.e., contiguous points) some delta below the threshold that would be interesting to examine further and **rank** them"

Segments = a series of consecutive time points
Ranking = some user defined priority of metrics (e.g., recency, segment length, avg value, etc)

Outline

Architecture

Dataset Generation

UI Implementation

Database Operator Implementation

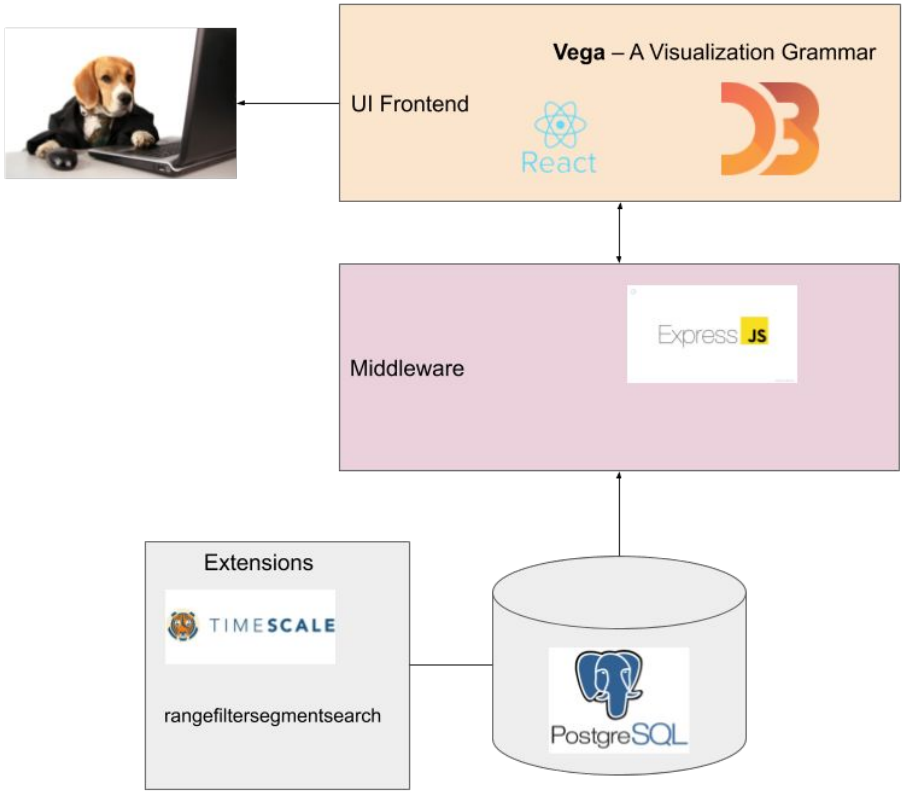
Performance Evaluation

Bugs

Future Work

Related Work

Overall Architecture



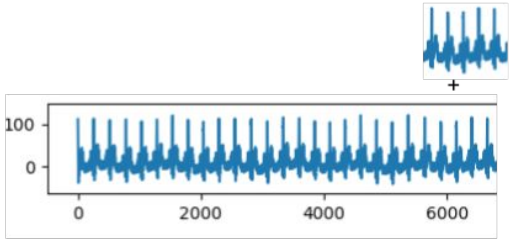
Dataset Generation

Want datasets with millions of points, several GB in size

No publicly available TS datasets of this size!

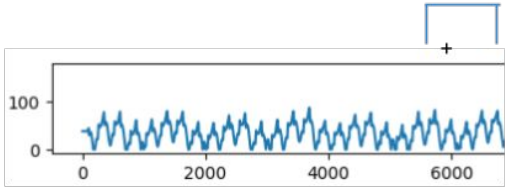
Used ECGSYN along with custom script to generate anomalies

Raw ECG Values



Superimpose time-shifted segment upon itself to generate anomalies

Detector Output Values



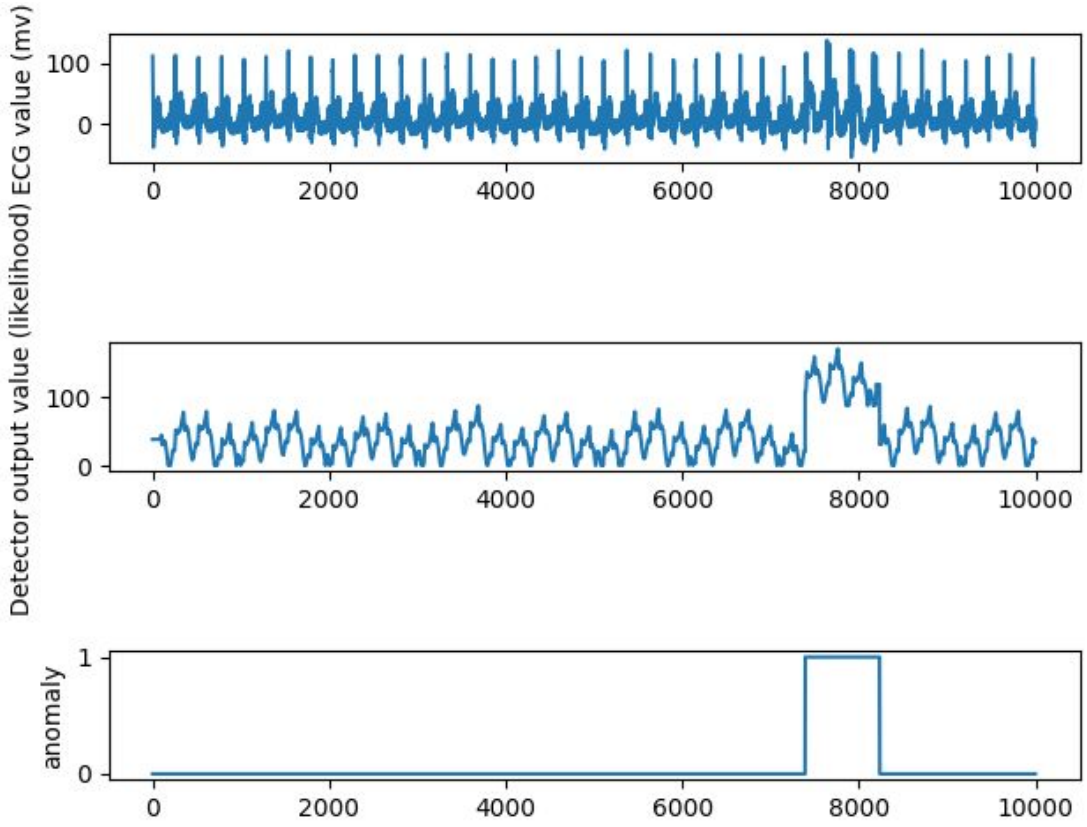
Use moving avg as detector output with some augmentation at anomalous ranges

Example Segment

Dataset Stats

3 chunks, each 1.1GB uncompressed (33M points). Largest data tested is **100M points**.

Spans 3 years, sampled each second uniformly



Demo

UI Implementation

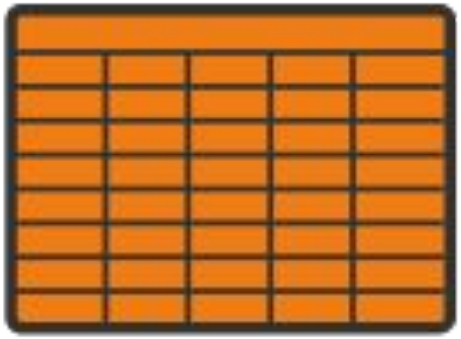
Loading all data into the chart at once is impractical

To address this, data is dynamically queried, loaded, and removed as the user pans and zooms through the chart

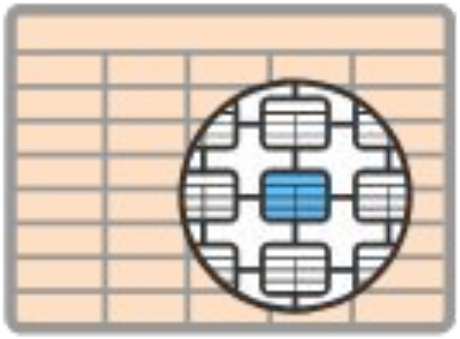
A buffer of data 2x the length of the current view is loaded when a retrigger is warranted

Trigger events are coming within 1 length of the beginning or end of the currently loaded data, or zooming out to a point where the view sees most of the currently loaded data

TimescaleDB: Architecture



Hypertable



Chunk

Abstraction over many individual tables holding data (chunks)

Hypertable partition by time and optionally by dimensions (partitioning across “time and space”)

Sized to ensure all B-trees for indexes can reside in memory during inserts and avoid expensive vacuuming ops

ECG Schema

```
CREATE TABLE IF NOT EXISTS "ecg_data"(
    ecg_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    ecg_mv NUMERIC NOT NULL,  --ecg reading in millivolts
    anomaly_likelihood NUMERIC
);

CREATE TABLE IF NOT EXISTS "ecg_data_with_lag"(
    ecg_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    ecg_datetime_prev TIMESTAMP WITHOUT TIME ZONE,  -- can be null
    ecg_mv NUMERIC NOT NULL,  --ecg reading in millivolts
    anomaly_likelihood NUMERIC
);
```

Hypertable defined on ecg_datetime column

B-tree index on ecg_mv column

B-tree index on anomaly_likelihood column

Database Operators

filter_segment(TABLE, …, min val, max val) → Returns filtered sub-relation with segment start times appended.

> Overloaded versions can take in additional params, like date filter range, prev_ts_col name.

window_autosample(TABLE,..., screen width, screen height) → Returns auto downsampled relation, where number of points returned is some multiple of screen width.

Performance Measurements

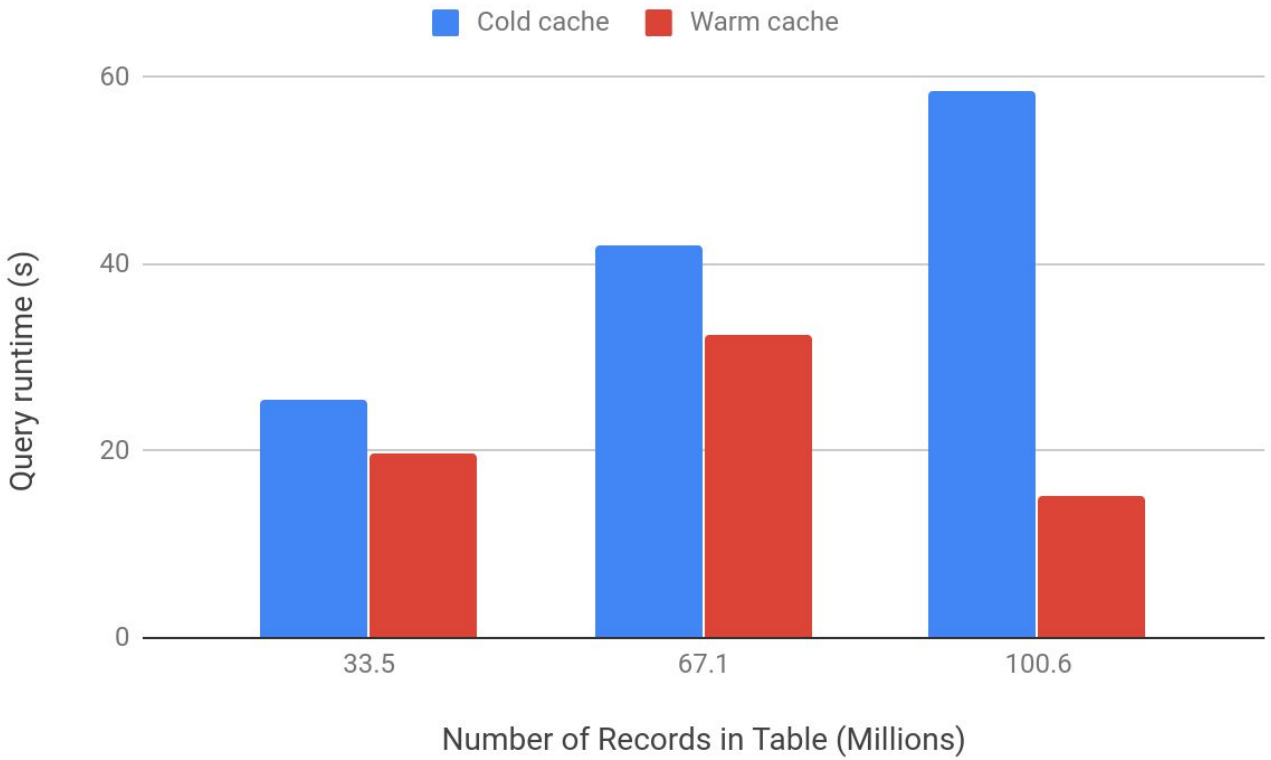
Setup used:

- 1 node
- 1 HDD (7200 RPM SATA)
- 8 core processor (AMD FX-8320E)
- 20GB RAM
- Ubuntu 16.10
- Postgres 10

Performance: Segment Filter

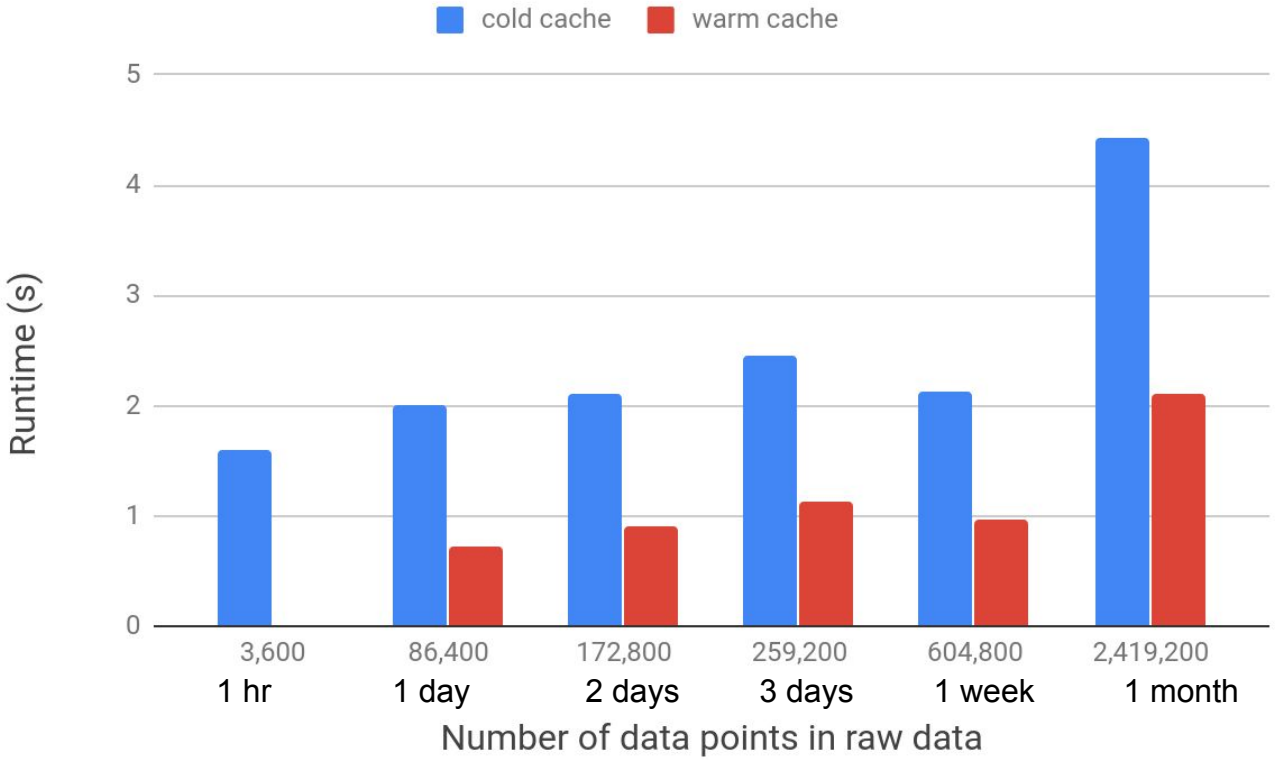
Filter criteria: all anomalous segments above threshold

Num of anomalous points:
33M: 660k
67M: 1.3M
100M: 2M



Performance: Window Autosample

Note: 1 point /s



Future opportunities

UI improvements (overlay anomalies, custom ranking UDFs, handling multi-channel TS data)

Optimizing segment_filter by converting to c instead of PL/pgsql

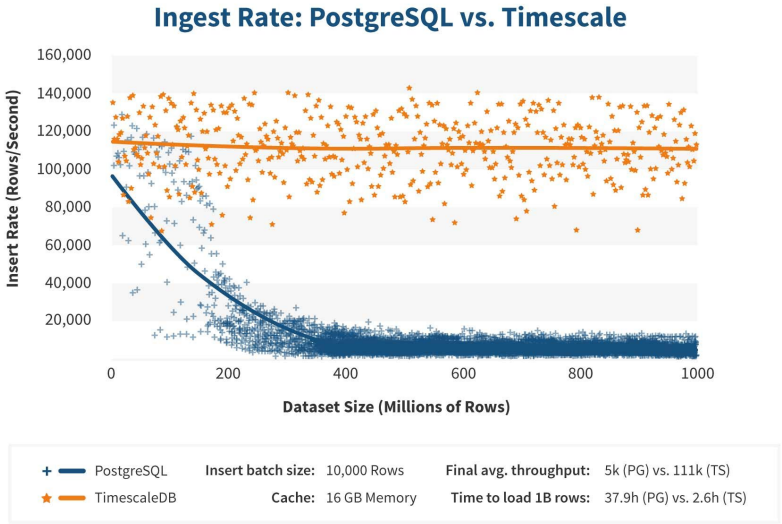
Many improvements to window_downsample (sampling strategies, indexing, approx querying)

Separating detector outputs into separate table and joining to raw values

Extra slides (for reference if needed)

TimescaleDB: Advantages vs PostgreSQL

Ingestion



Queries

More performant on:

- Time-based GROUP BYs
- Queries using time ordering: "Merge append" optimization

Similar perf on other queries

Supports full SQL and adds some custom functions

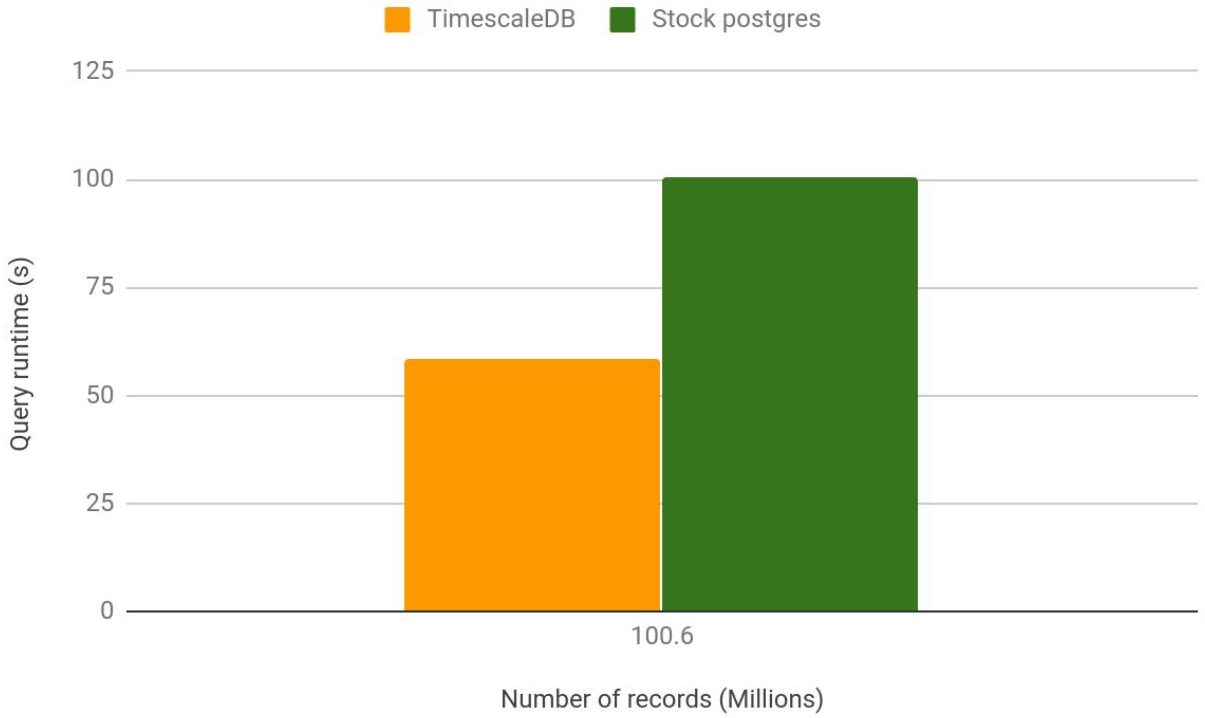
*Note: claimed by Timescale. We show some of our measurements later,

Segment Filter Perf vs Standard PostgreSQL

Filter criteria: all anomalous segments (above threshold)

Num of anomalous points:
100M: 2M

Cold cache



Ingestion: Timescale vs standard PostgreSQL

Benchmark queries

Segment filter

```
SELECT segment_start_ts AS start_date, COUNT(*) as number_points,
json_agg(value_to_passthru ORDER BY cur_ts) AS ecg_mv
FROM filter_segments(NULL::public.ecg_data_with_lag,
                'ecg_datetime', 'ecg_datetime_prev',
                'anomaly_likelihood', 'ecg_mv', 120, 423)
GROUP BY segment_start_ts
ORDER BY number_points DESC
LIMIT 10;
```

window_autosample

```
SELECT * FROM window_autosample(NULL::ecg_data100,
'ecg_datetime',
'ecg_mv', '2018-02-01 12:00:00', '2018-02-01 13:00:00', 1000,
300);
```

Segment Filter Algo

```
If previous timestamp column is not provided:
    Determine previous timestamps
Filter by date and detector output range criteria
Order results by timestamp

# TS of the previous entry evaluated in the loop
loop_prev_ts = null

Loop through results:
  Each entry has a (cur_ts, prev_ts)
   If prev_timestamp is null:
      Segment_start_ts = cur_ts
      cur_prev_ts = cur_ts
   Else:
      If loop_prev_ts is null:
        Segment_start_ts = cur_ts
        Loop_prev_ts = cur_ts
      Else if prev_ts = loop_prev_ts:
        # Is a continuation of segment
        Loop_previous_ts = cur_ts
      Else:
        # Is a new segment
        Segment_start_ts = cur_ts
        Loop_previous_ts = cur_ts
    Return tuple with segment_start_ts appended
```

Window Autosample Algo

```
Width_factor = 4

Max_samples = width_factor * screen_width

# Execute a count to get number of records in time range
Count = SELECT COUNT(*) from time range

If count <= max_samples:
    Return original relation without downsampling

Timespan_in_sec = date_max - date_min
Bucketsize_in_secs = int(
      timespan_in_sec / max_samples)

Return relation time bucketed and grouped by bucketsize
```